Outline – Beyond Classical Search

- Informed Searches
  - Best-first search
  - Greedy best-first search
  - A* search
  - Heuristics

- Local search algorithms
  - Hill-climbing search
  - Simulated annealing search
  - Local beam search

- Genetic algorithms

Chapter 4

Review: Tree search

- A search strategy is defined by picking the order of node expansion
  - Breadth-first
  - Depth-first
  - Iterative deepening

Implementation of search algorithms

Function General-Search(problem, Queuing-Fn) returns a solution, or failure

nodes ← make-queue(make-node(initial-state[problem]))

loop do
  if node is empty then return failure
  node ← Remove-Front(nodes)
  if Goal-Test[problem] applied to State(node) succeeds then return node
  nodes ← Queuing-Fn(nodes, Expand(node, Operators[problem]))
end

Queuing-Fn(queue, elements) is a queuing function that inserts a set of elements into the queue and determines the order of node expansion. Varieties of the queuing function produce varieties of the search algorithm.

Breadth First Search

Intuition: Expand all nodes at depth $i$ before expanding nodes at depth $i + 1$

- Complete? Yes.
- Optimal? Yes, if path cost is nondecreasing function of depth
- Time Complexity: $O(b^d)$
- Space Complexity: $O(b^d)$, note that every node in the fringe is kept in the queue.

Uniform Cost Search

Intuition: Expand the cheapest node. Where the cost is the path cost $g(n)$

- Complete? Yes.
- Optimal? Yes, if path cost is nondecreasing function of depth
- Time Complexity: $O(b^d)$
- Space Complexity: $O(b^d)$, note that every node in the fringe keep in the queue.

Note that Breadth First search can be seen as a special case of Uniform Cost Search, where the path cost is just the depth.
Artificial Intelligence

Depth First Search

- Enqueue nodes in LIFO (last-in, first-out) order.
- Complete? No (Yes on finite trees, with no loops).
- Optimal? No
- Time Complexity: $O(b^m)$, where $m$ is the maximum depth.
- Space Complexity: $O(bm)$, where $m$ is the maximum depth.

Intuition: Expand node at the deepest level (breaking ties left to right)

Depth-Limited Search

- Enqueue nodes in LIFO (last-in, first-out) order. But limit depth to $L$.
- Complete? Yes if there is a goal state at a depth less than $L$.
- Optimal? No
- Time Complexity: $O(b^L)$, where $L$ is the cutoff.
- Space Complexity: $O(bL)$, where $L$ is the cutoff.

Intuition: Expand node at the deepest level, but limit depth to $L$

Iterative Deepening Search I

- Do depth limited search starting at $L = 0$, keep incrementing $L$ by 1.
- Complete? Yes
- Optimal? Yes
- Time Complexity: $O(b^d)$, where $d$ is the depth of the solution.
- Space Complexity: $O(bd)$, where $d$ is the depth of the solution.

Intuition: Combine the Optimality and completeness of Breadth first search, with the low space complexity of Depth first search

Iterative Deepening Search II

- Iterative deepening looks wasteful because we re-explore parts of the search space many times.

Consider a problem with a branching factor of 10 and a solution at depth 5...

$1+10+100+1000+10,000+100,000 = 111,111$

$1+10$
$1+10+100$
$1+10+100+1000$
$1+10+100+1000+10,000$
$1+10+100+1000+10,000+100,000$

$= 123,456$

Example - Problem Design

- State-space search problem
  - World can be in one of several states
  - Moves can change the state of the world
- Design/create data structures to represent the “world”
- Design/create functions representing “moves” in the world

Example: Farmer, Wolf, Goat, and Cabbage Problem

- A farmer with his wolf, goat, and cabbage come to the edge of a river they wish to cross.
- There is a boat at the river's edge, but, of course, only the farmer can row.
- The boat can carry a maximum of the farmer and one other animal/vegetable.
- If the wolf is ever left alone with the goat, the wolf will eat the goat.
- Similarly, if the goat is left alone with the cabbage, the goat will eat the cabbage.
- Devise a sequence of crossings of the river so that all four of them arrive safely on the other side of the river.
Code in sssearch directory

- http://www2.hawaii.edu/~nreed/lisp/sssearch/

FWGC State Creation and Access

- Functions to create and access states
  ;; Use a list to represent the state of the world -
  ;; i.e. the side of the river that each of the 4
  ;; characters is on. East and West are the sides.
  (defun make-state (f w g c) (list f w g c))
  (defun farmer-side (state) (nth 0 state)) ; or (car state)
  (defun wolf-side (state) (nth 1 state)) ; or (cadr state)
  (defun goat-side (state) (nth 2 state)) ; or (caddr state)
  (defun cabbage-side (state) (nth 3 state)) ; or (cadddr state)

FWGC Start, Goal and Moves

- Global variables
  ;; Represent start and goal states and list
  ;; of possible moves
  ; Start state *start*
  (setq *start* (make-state 'e 'e 'e 'e))
  ; Goal state *goal*
  (setq *goal* (make-state 'w 'w 'w 'w))
  ; Possible moves *moves*
  (setq *moves* '(farmer-takes-self farmer-takes-wolf
  farmer-takes-goat farmer-takes-cabbage))

Utility Functions

- Determine the opposite side and
  if a state is safe
  ;; Note – these are very simple,
  ;; however they make it easy to
  ;; change – e.g. north and south
  ;; instead of east and west
  (defun opposite (side) ; Return

Move Function Code

(defun farmer-takes-self (state) (cond
  ((safe (make-state (opposite (farmer-side state)))
    ; if safe, move
    (wolf-side state)
    (goat-side state)
    (cabbage-side state)))); return new state
  (t nil))); otherwise - return F

(defun farmer-takes-wolf (state) (cond
  ((equal (farmer-side state) (wolf-side state))
    ; if safe, move
    (safe (make-state (opposite (farmer-side state))
    (goat-side state)
    (cabbage-side state)))); return new state
  (t nil))); otherwise - return F

This means that everybody/everything is on the same side of the river.

This means that we somehow got the Wolf to the other side.
Summary – Uninformed Search

The search techniques we have seen so far...

- Breadth first search
- Uniform cost search
- Depth first search
- Depth limited search
- Iterative Deepening

...are all too slow for most real world problems
Heuristic Search (informed)

• A **Heuristic** is a function that, when applied to a state, returns a number that is an estimate of the merit of the state, with respect to the goal. We can use this knowledge of the relative merit of states to guide search.

• In other words, the heuristic tells us approximately how far the state is from the goal state*. But for reasons which we will see, heuristics that only underestimate are very desirable, and are called admissible.

*Le Smaller numbers are better

---

Best-first search

- Idea: use an **evaluation function** \( f(n) \) for each node
  - estimate of "desirability" Expand most desirable unexpanded node
- **Implementation:**
  Order the nodes in open list (fringe) in decreasing order of desirability
- **Special cases:**
  - greedy best-first search
  - A* search

---

Greedy best-first search

- **Evaluation function** \( f(n) = h(n) \) (heuristic)
  - = estimate of cost from \( n \) to goal
- **e.g.**, \( h_{SLO}(n) \) = straight-line distance from \( n \) to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal

---

Example: Romania

- You’re in Romania, on vacation in Arad.
- Flight leaves tomorrow from Bucharest.

- Formulate goal: be in Bucharest
- Formulate problem: states: various cities
- operators: drive between cities
- Find solution: sequence of cities, such that total driving distance is minimized.
Romania Straight-Line Distances to Bucharest

<table>
<thead>
<tr>
<th>City</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>366</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Craiova</td>
<td>160</td>
</tr>
<tr>
<td>Dobroța</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Făgăraș</td>
<td>176</td>
</tr>
<tr>
<td>Giurgiu</td>
<td>77</td>
</tr>
<tr>
<td>Iași</td>
<td>226</td>
</tr>
<tr>
<td>Lugoj</td>
<td>244</td>
</tr>
<tr>
<td>Mădăia</td>
<td>241</td>
</tr>
<tr>
<td>Neamț</td>
<td>234</td>
</tr>
<tr>
<td>Oradea</td>
<td>380</td>
</tr>
<tr>
<td>Pitești</td>
<td>100</td>
</tr>
<tr>
<td>Rimnicu Vîlcea</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Timișoara</td>
<td>329</td>
</tr>
<tr>
<td>Urziceni</td>
<td>50</td>
</tr>
<tr>
<td>Vaslui</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>

**Greedy best-first search example**

**Properties of Greedy Best-First Search**

- **Complete?** No – can get stuck in loops, e.g., Iași → Neamț → Iași → Neamț →
- **Time?** $O(b^m)$, but a good heuristic can give dramatic improvement
- **Space?** $O(b^m)$ -- keeps all nodes in memory
- **Optimal?** No
The A* Search Algorithm (“A-Star”)

- Idea: avoid expanding paths that are already expensive
- Keep track of a) the cost to get to the node as well as b) the cost to the goal
- Evaluation function \( f(n) = g(n) + h(n) \)
  - \( g(n) \) = cost so far to reach \( n \)
  - \( h(n) \) = estimated cost from \( n \) to goal
  - \( f(n) \) = estimated total cost of path through \( n \) to goal

The A* Algorithm

\( f(n) \) as the estimated cost of the cheapest solution that goes through node \( n \)

Use the general search algorithm with a priority queue queuing strategy.

If the heuristic is optimistic, that is to say, it never overestimates the distance to the goal, then...

A* is optimal and complete!
Example: 8-puzzle

- State: integers on each tile
- Operators: moving blank left, right, up, down (move must be possible)
- Goal test: does state match goal state?
- Path cost: 1 per move
Encapsulating state information in nodes

A state is a (representation of) a physical configuration. A node is a data structure constituting part of a search tree, includes parent, children, depth, path cost g(χ). States do not have parents, children, depth, or path cost!

The Expand function creates new nodes, filling in the various fields and using the Operators (or SuccessorFn) of the problem to create the corresponding states.

Heuristic Evaluation Function #1

;;;; HEURISTIC-EVAL-1 - Return the sum of the distances tiles are out of place.
;;;; A better heuristic for the n-puzzle problem than # tiles out of place.

(defun heuristic-eval-1 (state goal)
  "Sum the distance for each tile from its goal position."
  (do ;; loop adding distance
      ((sum 0) (tile 1 (+ tile 1))) ;; for tiles 1 to (n^2 - 1)
      ((equal tile (* n n)) sum) ;; return sum when done
      (setq sum (+ sum (distance tile state goal)))))

Heuristic Evaluation Function #2

;;;; HEURISTIC-EVAL-2 - Return the sum of the distances out of place plus two times the number of direct tile reversals.
(defun heuristic-eval-2 (state goal)
  "Sum of distances plus 2 times the number of tile reversals"  
  (+ (heuristic-eval-1 state goal) (* 2 (tile-reversals state goal))
    (heuristic-eval-2 state goal))

(defun tile-reversals (state goal)
  "Calculate the number of tile reversals between two states"
  (cond ;; if one of the tiles is the blank, it doesn’t count
        ((or (equal (nth pos1 state1) *blank*)           
             (equal (nth pos2 state1) *blank*)
           ) 0) ;; return 0
        ((and                           
           (equal (nth pos1 state1) (nth pos2 state2))
           (equal (nth pos2 state1) (nth pos1 state2)))
         1) ;; return 1
        (t 0))

(defun tile-reverse (pos1 pos2 state1 state2)
  "Return 1 if the tiles in the two positions are reversed, else 0"  
  (cond (or (equal (nth pos1 state1) *blank*)         
            (equal (nth pos2 state1) *blank*)) ; blanks don’t count
        0) ;; return 0
        (and ; the tiles are reversed
            (equal (nth pos1 state1) (nth pos2 state2))
            (equal (nth pos2 state1) (nth pos1 state2))
          ) 1) ;; return 1
        (t 0)) ; else return 0
Distance Function

;; DISTANCE - calculate the distance a tile is from
;; its goal position.
(defun distance (tile state goal)
  "Calculate the Manhattan distance a tile is from its
  goal position."
  (+
    (abs (- (row tile state)
            (row tile goal)))
    (abs (- (column tile state)
            (column tile goal))))

Admissible Heuristics

- A heuristic \( h(n) \) is admissible if for every node \( n \),
  \( h(n) \leq h^*(n) \), where \( h^*(n) \) is the true cost to reach
  the goal state from \( n \).
- An admissible heuristic never overestimates the
  cost to reach the goal, i.e., it is optimistic
- Example: \( h_{SLD}(n) \) (never overestimates the actual
  road distance)
- Theorem: If \( h(n) \) is admissible, \( A^* \) using TREE-
  SEARCH is optimal

Consistent Heuristics

- A heuristic is consistent if for every node \( n \), every successor
  \( n' \) of \( n \) generated by any action \( a \),
  \( h(n) \leq c(n,a,n') + h(n') \)
- If \( h \) is consistent, we have
  \[ f(n') = g(n') + h(n') \]
  \[ \geq g(n) + c(n,a,n') + h(n') \]
  \[ = g(n) + h(n) \]
  \[ = f(n) \]
- i.e., \( f(n) \) is non-decreasing along any path.
- Theorem: If \( h(n) \) is consistent, \( A^* \) using GRAPH-
  SEARCH is optimal

Optimality of \( A^* \)

- \( A^* \) expands nodes in order of increasing \( f \) value
- Gradually adds "\( f \)-contours" of nodes
- Contour \( i \) has all nodes with \( f = f_i \), where \( f_i < f_{i+1} \)

Optimality of \( A^* \) (proof)

- Suppose some suboptimal goal \( G_2 \) has been generated and is
  in the fringe. Let \( n \) be an unexpanded node in the fringe such
  that \( n \) is on a shortest path to an optimal goal \( G \).
- \( f(G_2) = g(G_2) \) since \( h(G_2) = 0 \)
- \( g(G_2) > g(G) \) since \( G_2 \) is suboptimal
- \( f(G) = g(G) \) since \( h(G) = 0 \)
- \( f(G_2) > f(G) \) from above
Artificial Intelligence

Optimality of A* (proof)
- Suppose some suboptimal goal \( G_2 \) has been generated and is in the fringe. Let \( n \) be an unexpanded node in the fringe such that \( n \) is on a shortest path to an optimal goal \( G \).

\[
\begin{align*}
\text{(1)} & \quad f(G_2) > f(G) & \text{from above} \\
\text{(2)} & \quad h(n) \leq h^*(n) & \text{since } h \text{ is admissible} \\
\text{(3)} & \quad g(n) + h(n) \leq g(n) + h^*(n) \\
\text{(4)} & \quad f(n) \leq f(G) \\
\end{align*}
\]

Hence \( f(G_2) > f(n) \), and A* will never select \( G_2 \) for expansion.

Properties of A*
- **Complete?** Yes (unless there are infinitely many nodes with \( f \leq f(G) \))
- **Time?** Exponential
- **Space?** Keeps all nodes in memory
- **Optimal?** Yes

Admissible heuristics
E.g., for the 8-puzzle:
- \( h_1(n) \) = number of misplaced tiles
- \( h_2(n) \) = total Manhattan distance (i.e., no. of squares from desired location of each tile)

\[
\begin{align*}
7 & 2 & 4 \\
5 & 6 & 3 \\
8 & 1 & 1
\end{align*}
\]

\[
\begin{align*}
1 & 2 & 4 \\
5 & 6 & 3 \\
8 & 1 & 1
\end{align*}
\]

\[
\begin{align*}
h_1(S) = 8 \\
h_2(S) = 3+1+2+2+2+3+3+2 = 18
\end{align*}
\]

Dominance
- If \( h_2(n) \geq h_1(n) \) for all \( n \) (both admissible)
- then \( h_2 \) dominates \( h_1 \)
- \( h_1 \) is better for search
- Typical search costs (average number of nodes expanded):
  - \( d=12 \):
    - IDS = 3,644,035 nodes
    - A*(\( h_1 \)) = 227 nodes
    - A*(\( h_2 \)) = 73 nodes
  - \( d=24 \):
    - IDS = too many nodes
    - A*(\( h_1 \)) = 39,135 nodes
    - A*(\( h_2 \)) = 1,641 nodes

Relaxed problems
- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then \( h_1(n) \) gives the shortest solution
- If the rules are relaxed so that a tile can move to any adjacent square, then \( h_2(n) \) gives the shortest solution
Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it

Example: n-queens

- Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Hill-Climbing Search

- "Like climbing Everest in thick fog with amnesia"

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
neighbor, a node

current = MAKE-NODE([INITIAL-STATE][problem])

loop do
    neighbor = a highest-valued successor of current
    if VALUE[neighbor] \leq VALUE[current] then return STATE[current]
    current = neighbor

```

Hill-Climbing Search: 8-Queens Problem

- $h$ = number of pairs of queens that are attacking each other, either directly or indirectly
- $h = 17$ for the above state

- A local minimum with $h = 1$
Simulated Annealing Search

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency

```python
function SIMULATED-ANNEALING(problem, schedule)
inputs: problem, a problem
        schedule, a mapping from time to temperature
local variables: current, a node
                next, a node
                T, a "temperature" controlling prob. of downward steps

current ← MAKE-INITIAL-STATE(problem)
for \( t \) in schedule do
    if \( t = 0 \) then return current
    next ← a randomly selected successor of current
    \( \Delta E \) ← VALUE(next) - VALUE(current)
    if \( \Delta E > 0 \) then current ← next
    else current ← next only with probability \( e^{\Delta E/T} \)
```

Properties of Simulated Annealing Search

- One can prove: If \( T \) decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
- Widely used in VLSI layout, airline scheduling, etc

Local Beam Search

- Keep track of \( k \) states rather than just one
- Start with \( k \) randomly generated states
- At each iteration, all the successors of all \( k \) states are generated
- If any one is a goal state, stop; else select the \( k \) best successors from the complete list and repeat.

Genetic Algorithms

- A successor state is generated by combining two parent states
- Start with \( k \) randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation

Genetic Algorithms

- Fitness function: number of non-attacking pairs of queens (min = 0, max = \( 8 \times 7/2 = 28 \))
- \( 24/(24+23+20+11) = 31\% \)
- \( 23/(24+23+20+11) = 29\% \) etc
Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Informed Searches
  - Best-first search
  - Greedy best-first search
  - A* search
  - Heuristics
- Local search algorithms
  - Hill-climbing search
  - Simulated annealing search
  - Local beam search
- Genetic algorithms
- Chapter 4